To Be Destructive or Not To Be, That is the Question on Modular Extensions

Shigeru Chiba
The University of Tokyo

Quotes

- Aspect oriented programming is quantification and obliviousness.
  - Robert E. Filman Daniel P. Friedman

- Obliviousness is not mandatory but desirable.
  - Awais Rashid?

AOP functionality (1)

- Obliviousness is useful and practical!
  - An advice can obliviously modify a method.
    - The original source code is not modified at all when the software is extended.

```
class AddExpr {
  Value eval() { … }
}
```

```
Value around(AddExpr ae):
    execution(Value AddExpr.eval())
    && this(ae) {
    if (…) proceed();
    else … ;
}
```

AOP functionality (2)

- Limited scope
  - An advice can modify a method call in a body
  - Breaking modularity?

```
class VarDecl {
  Value init() {
    v = right.eval();
  } }

      class AddExpr {
        Value eval() { ... }
      }
```

```
aspect Logging {
  before():
    call(void Expr.eval())
    && withincode(* VarDecl.init()) {
    ...
  } }
```

A new scripting language in two weeks

amazon.co.jp

Your Store | Amazon Points | Gift Cards | Today's Deals | Sell | Help | 日本語

日産 デイズ ルークス
抽選でAmazonギフト券が当たる 今すぐチェック

Shop by Department | Japanese Books | Go | Hello. Sign in Your Account | Join Prime | 0 Cart | Wish List

Japanese Books | Advanced Search | Browse Genres | New & Future Release | Amazon Ranking | Comics | Magazines | Bunko & Shinsho | Amazon Student | Bargain Books

クリック なか見! 検索

Share your own customer images

Search inside this book

Tell the Publisher!
I'd like to read this book on Kindle

2週間でできる! スクリプト言語の作り方 (Software Design plus) [単行本（ソフトカバー）]

千葉 滋 (著)

★★★★☆ (5 customer reviews)

Price: ￥2,786 & eligible for **Free Shipping**. Details

Only 12 left in stock (more on the way). Click here for details of availability.
Ships from and sold by Amazon.co.jp. Gift-wrap available.

住所からお届け予定日を確認 113-0033 - 東京都文京区本郷 詳細

本日, 4月 17 にお届けするには、今から 6 時間 と 59 分以内に「お急ぎ便」または「当日お急ぎ便」を選択して注文を確定してください（有料オプション。 Amazonプライム会員 は無料）

4 used from ￥1,973

発表!オールタイムベスト コミック100
オールタイムベスト第2弾 「みんなにすすめたいコミック100」 。LINEアンケート結果も発表!無料の『オールタイムベストコミック100 [Kindle版]』 配布中。
› See more product promotions

このページを日本語で表示しますか?
ここをクリック

Quantity: 1

Add to Shopping Cart

or

Sign in to turn on 1-Click ordering.

or

Add to Cart with Free Expedited Shipping

With an Amazon Prime membership. Sign up when you check out. Learn More

Add to Wish List

More Buying Choices

5 used & new from ￥1,973

Have one to sell? Sell yours here

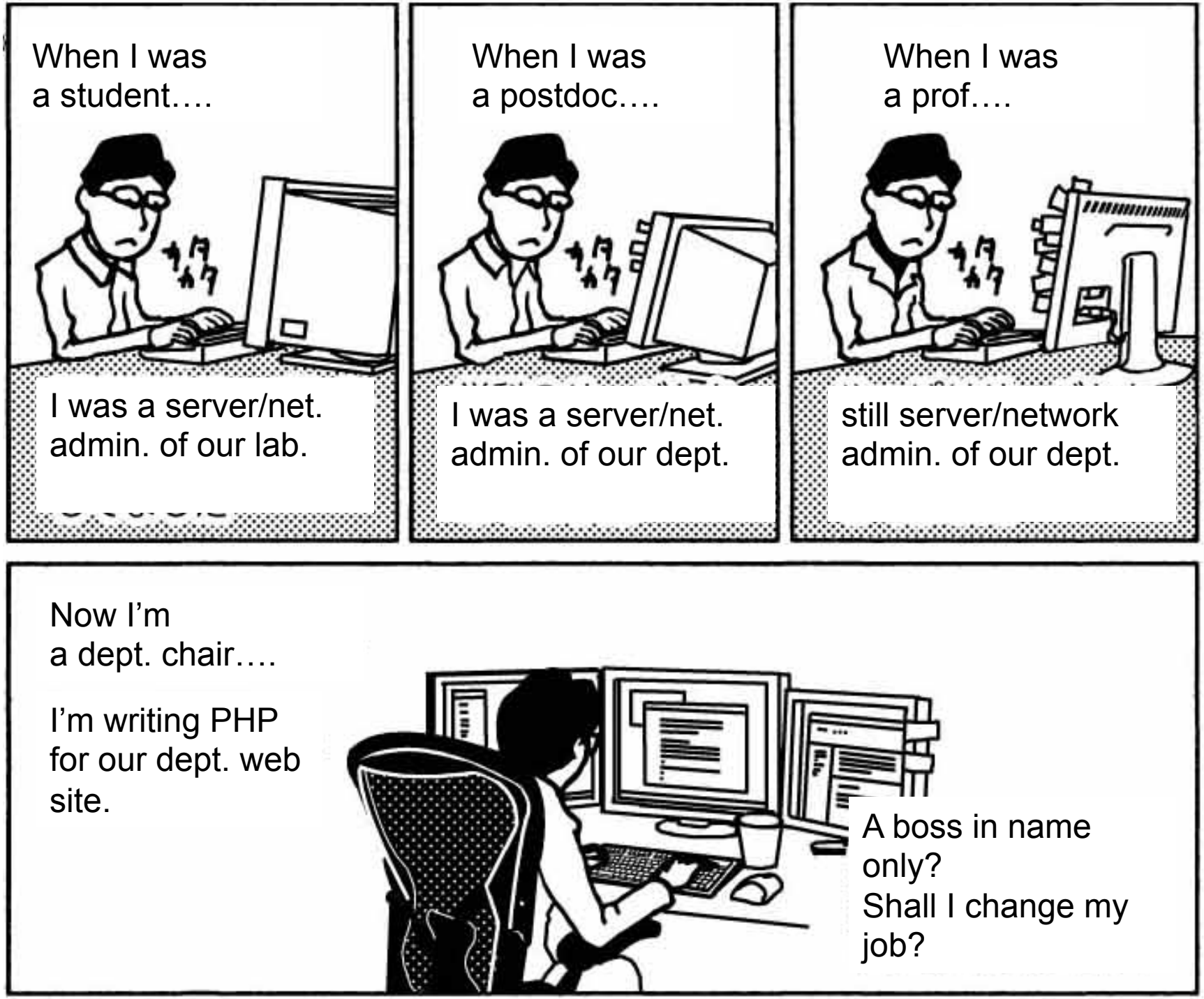
Share

Shiger

http://www.amazon.co.jp/gp/switch-language/product/4774149748/ref=dp_change_lang?ie=UTF8&language=en_JP

A new scripting language in two weeks

http://www.amazon.co.jp/gp/switch-language/product/4774149748/ref=dp_change_lang?ie=UTF8&language=en_JP



S. Chiba, A new scripting language in two weeks, Gihyo co., 2012

Éric Tanter said to me.

- If the code in the book is in Scheme, you don't need obliviousness or AOP.

- … Right. But Scheme also provides "obliviousness"
  or destructive extension I call.

  – The code in my book is in Java.

[Chiba,Igarashi,Zakirov, OOPSLA'10]

GluonJ: A reviser

- Destructive extension
  - A reviser can add and ~~override~~ modify a method, and add a field to an existing class.
    - It cannot have an explicit constructor.

```
class AddExpr {
  Value eval() { ... }
}
```

```
class FloatEx revises AddExpr {
  Value eval() {
    if (...) super.eval();
    else ... ;
  }
}
```


Shigeru Chiba, The University of Tokyo

GluonJ: A within method

- Limited scope
  - A method may have a predicate.
  - Its method overriding is effective only when it is called from …

```
class VarDecl {
  Value init() {
    v = right.eval();
  } }

      class AddExpr {
        Value eval() { … }
      }
```

```
class Log revises AddExpr {
    Value eval()
                within VarDecl.init() {
        …
    } }
```

Contextual predicate dispatch

- GluonJ
  - Predicates refers to non-local contexts i.e. **within** who is a caller.
    - Currently only **within** is available.
  - to deal with crosscutting concerns

- Original predicate dispatch
  - Predicates refers to only local contexts such as arguments and receiver's fields
    - for unambiguity and exhaustiveness

Shigeru Chiba, The University of Tokyo

Subclassing, mixin, traits, …

- Non-destructive extension
  - Both the original and the extension coexist.
    - The source code is not modified as in AOP.

```
class AddExpr {
  Value eval() { … }
}

AddExpr e1, e2;
e1 = new AddExpr();
e2 = new FloatEx();
```

```
class FloatEx extends AddExpr {
  Value eval() {
    if (…) super.eval();
    else … ;
  }
}
```


Shigeru Chiba, The University of Tokyo

To Be Destructive or Not To Be, That is the Question on Modular Extensions

Abstract Factory pattern or dependency injection

- AOP-like modification by non-destructive extension (= subclassing)

```
class Factory {
  AddExpr makeAddExpr() {
     return new AddExpr();
  } }

class FactoryEx extends Factory {
  AddExpr makeAddExpr() {
     return new FloatEx();
  } }

AddExpr e
  = factory.makeAddExpr();
```

```
class AddExpr {
  Value eval() { … }
}

class FloatEx extends AddExpr {
  Value eval() {
    if (…) super.eval();
    else … ;
  }
}
```

Abstract Factory pattern or dependency injection

- To switch classes,
  the main method must be modified by hand.

```
void main(String[] args) {
  factory = new FactoryEx();

  program.start(args);
}
```

Or, another main method must be written from scratch.

Abstract Factory pattern or dependency injection

- To switch classes,
  the main method must be modified by hand.

```
void main(String[] args) {
  factory = new FactoryEx();

  program.start(args);
}
```

```
void main(String[] args) {
  factory = new Factory();

  program.start(args);
}
```

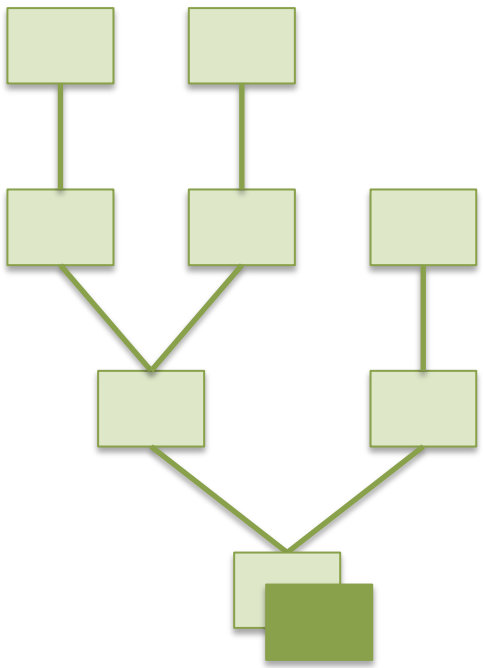
Or, another main method must be written from scratch.

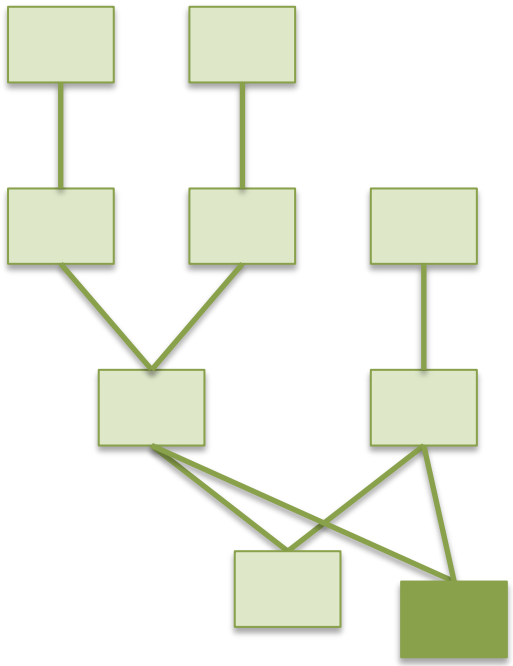
Also,

- Family polymorphism/virtual classes
  - are non-destructive
    like Abstract Factory pattern.

Destructive or Non-destructive

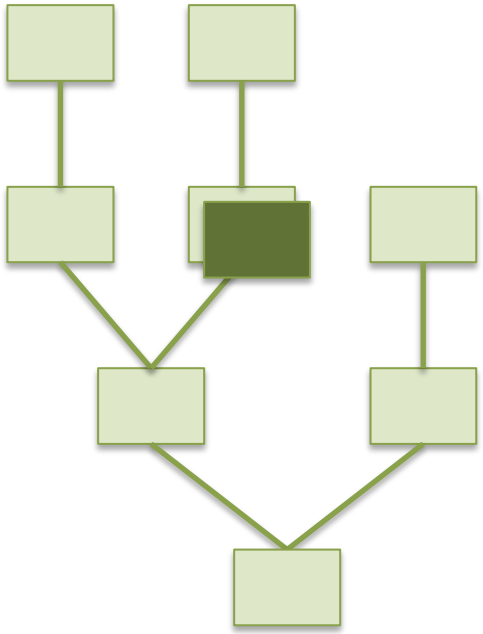
- Modification or Another copy

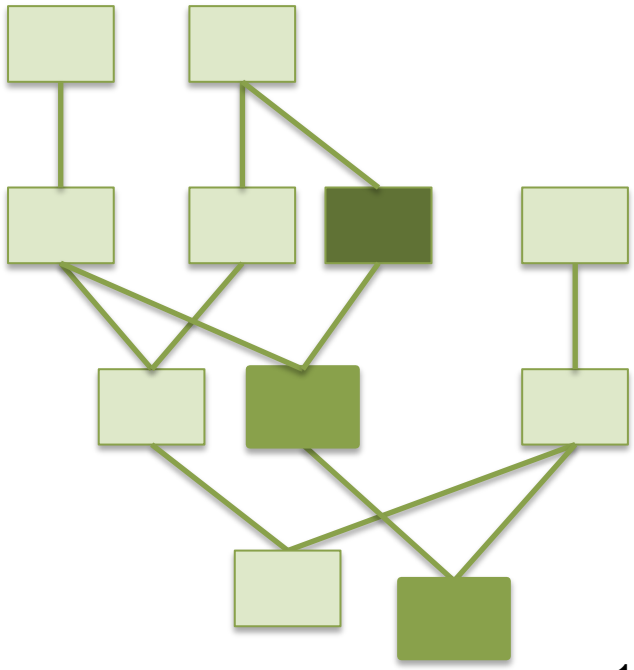




Destructive or Non-destructive

- Modification or Another copy
  - when an intermediate module is modified





Shigeru Chiba, The University of Tokyo

Destructive extension

- OK, it's useful when I want to modify only a piece of code in my program.

- But, I often want to reuse the original code in the same program.

Scope!

- Destructive — Always modify
- Conditionally
  - AspectJ's within, withincode, and cflow
  - GluonJ's within
  - ContextJ

    :
- Non-destructive — Only specific instances

Shigeru Chiba, The University of Tokyo

Various kinds of scopes

- In Ruby

```
class Integer
  def div(x)
      # returns a
      # rational num.
  end
end
```

destructive

```
class Integer
  def div(x)
      # returns an
      # integer result
  end
end
```

include

```
def average(list)
   sum = ...
   sum / list.size
end
```

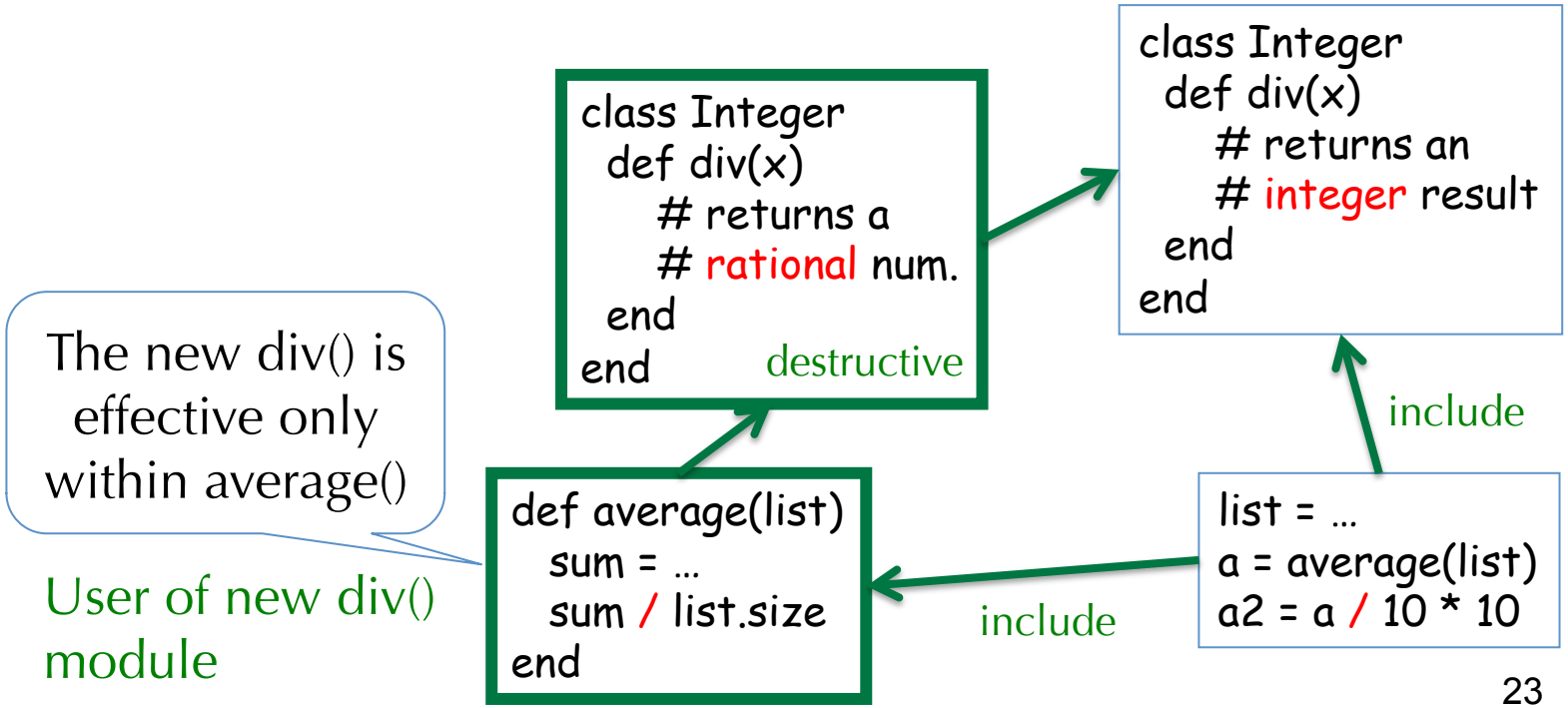
include

```
list = ...
a = average(list)
a2 = a / 10 * 10
```


Shigeru Chiba, The University of Tokyo

Reusable destructive extensions

- Module users should specify where they are effective.
  - Module writers should not.



Mentioning the scope

- At the side of the module user.

- AspectJ
  - abstract pointcut

- Dynamic Aspect-Oriented Programming
  - deploy(…) { … } in CaesarJ

- Context-Oriented Programming
  - with(…) { … } in ContextJ

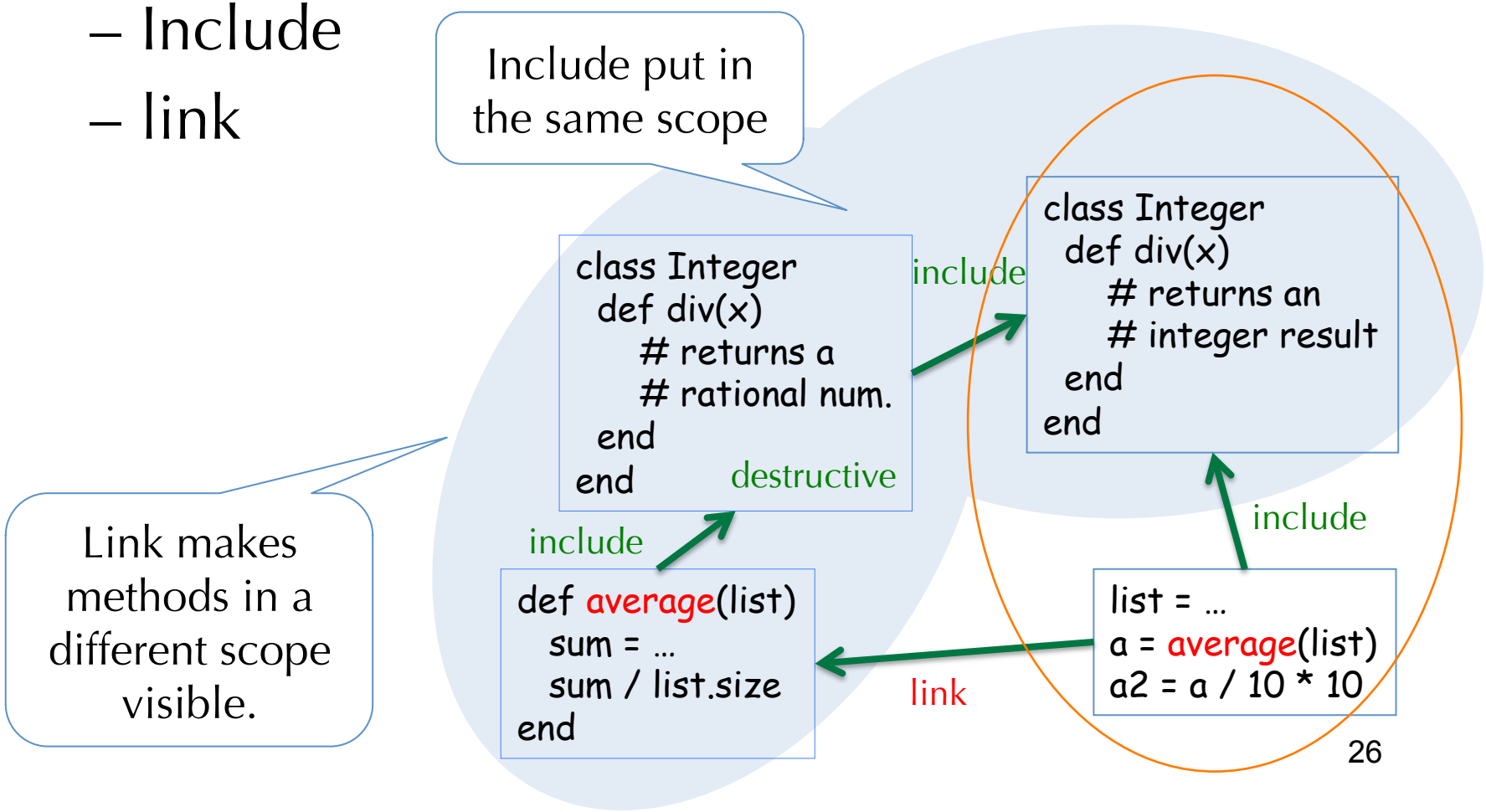
More structural scope

- Method shelters [Akai&Chiba, AOSD'12]
- Method shells [Takeshita&Chiba, SC'13]

- When a ``module" is imported,
  - the scope of the destructive extensions in it is declaratively specified.

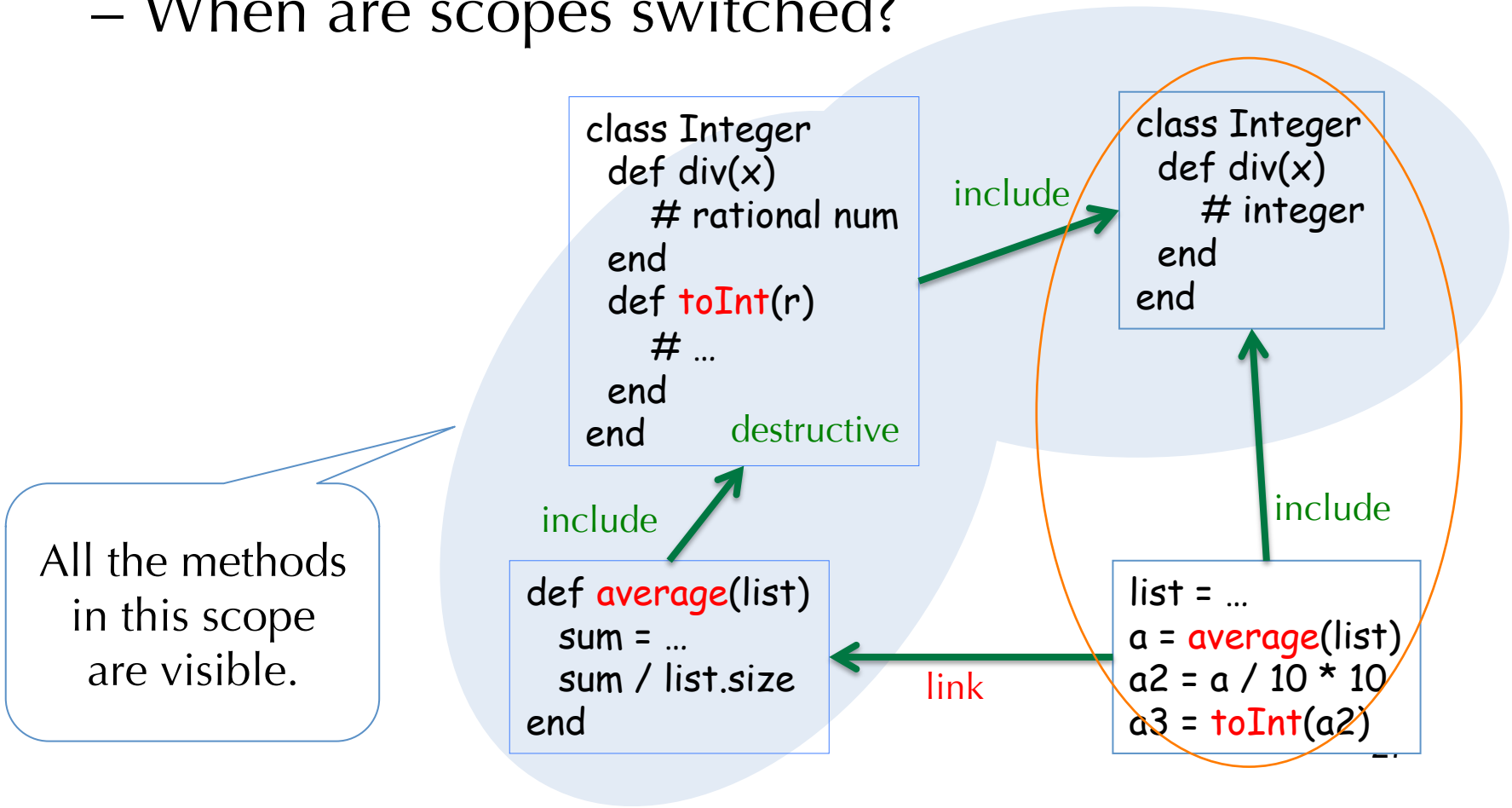
Method shells [Takeshita&Chiba'13]

- Two kinds of module import
  - Include
  - link



The semantics of link is comlex

- Takeshita's master thesis in 2014
  - When are scopes switched?



Summary

- To Be Destructive or Not To Be, That is the Question on Modular Extensions

- Destructive extensions Always

- Conditionally ...
  - Specified by extension-users
  - Structural scope e.g. Method Shelters/Shells

- Non-destructive Only specific instances